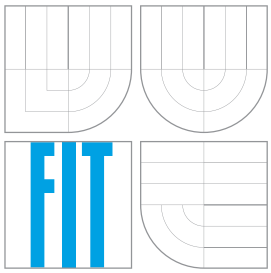
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVĚ AKCELEROVANÁ FUNKČNÍ VERIFIKACE

HARDWARE ACCELERATED FUNCTIONAL VERIFICATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARCELA ŠIMKOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL KAJAN

BRNO 2011

Abstrakt

Funkční verifikace je jednou z nejrozšířenějších technik ověřování korektnosti hardwarových systémů podle jejich specifikace. S nárůstem složitosti současných systémů se zvyšují i časové požadavky kladené na funkční verifikaci, a proto je důležité hledat nové techniky urychlení tohoto procesu. Teoretická část této práce popisuje základní principy různých verifikačních technik jako jsou simulace a testování, funkční verifikace, jakož i formální analýzy a verifikace. Následuje popis tvorby verifikačních prostředí nad hardwarovými komponentami v jazyce SystemVerilog. Část věnující se analýze popisuje požadavky kladené na systém pro akceleraci funkční verifikace, z nichž nejdůležitější jsou možnost jednoduchého spuštění akcelerované verze verifikace a časová ekvivalence akcelerovaného a neakcelerovaného běhu verifikace. Práce dále představuje návrh verifikačního rámce používajícího pro akceleraci běhů verifikace technologii programovatelných hradlových polí se zachováním možnosti spuštění běhu verifikace v uživatelsky přívětivém ladicím prostředí simulátoru. Dle experimentů provedených na prototypové implementaci je dosažené zrychlení úměrné počtu ověřovaných transakcí a komplexnosti verifikovaného systému, přičemž nejvyšší zrychlení dosažené v sadě experimentů je více než 130násobné.

Abstract

Functional verification is a widespread technique to check whether a hardware system satisfies a given correctness specification. The complexity of modern computer systems is rapidly rising and the verification process takes a significant amount of time. It is a challenging task to find appropriate acceleration techniques for this process. In this thesis, we describe theoretical principles of different verification approaches such as simulation and testing, functional verification, and formal analysis and verification. In particular, we focus on creating verification environments in the SystemVerilog language. The analysis part describes the requirements on a system for acceleration of functional verification, the most important being the option to easily enable acceleration and time equivalence of an accelerated and a non-accelerated run of a verification. The thesis further introduces a design of a verification framework that exploits the field-programmable gate array technology, while retaining the possibility to run verification in the user-friendly debugging environment of a simulator. According to the experiments carried out on a prototype implementation, the achieved acceleration is proportional to the number of checked transactions and the complexity of the verified system. The maximum acceleration achieved on the set of experiments was over 130 times.

Klíčová slova

funkční verifikace, testovací prostředí, SystemVerilog, hardwarová akcelerace, FPGA

Keywords

functional verification, testbench, SystemVerilog, hardware acceleration, FPGA

Citace

Marcela Šimková: Hardware Accelerated Functional Verification, diplomová práce, Brno, FIT VUT v Brně, 2011

Hardware Accelerated Functional Verification

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana Ing. Michala Kajana a uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Marcela Šimková
25. května 2011

Poděkování

Především bych chtěla poděkovat vedoucímu mé diplomové práce Ing. Michalu Kajanovi a mému konzultantovi Ing. Ondřeji Lengálovi za odborné vedení, ochotu při konzultacích této práce, konstruktivní kritiku a morální podporu.

Contents

Chapter 1

Introduction

Computer systems play nowadays very important part in human everyday lives. They are everywhere and they help people in many ways: they assist them in their working lives, they help them in their households and they even entertain them. To do this job precisely, the most important property of these systems is their correctness with respect to their specification (i.e. ensuring that they do not contain faulty behaviour). However, the complexity of modern computer systems is rising rapidly so achieving high degree of correctness is a difficult challenge. The discipline dealing with this issue is called *verification*. It has been observed that verification becomes the major bottleneck in the development of hardware systems, as it takes up to 80 % of the overall development cost and time [19]. For verification of computer hardware a variety of options is available to engineers: (*i*) simulation and testing, (*ii*) functional verification, and (*iii*) formal analysis and verification.

Although simulation and testing might be seen as old-fashioned methods, they are still highly effective especially in the early phase of implementation and debugging of base system functions. The benefits of testing real hardware (either in the form of an *application-specific integrated circuit* (ASIC) or a configuration of a *field-programmable gate array* (FPGA)) is the speed of testing (as it is performed in real time) and also the possibility to cover faults arising from the technology used for physical implementation of the logical circuit. Software simulation of hardware allows the developer to check that base system functions conform to system specification even before the circuit is physically assembled.

A more efficient approach to verification of systems is functional verification. This technique is based on a simulation of the environment of the system, the system being called a *design under test* (DUT), and uses *coverage driven verification*, *constrained-random stimulus generation*, *assertion-based verification*, and other methods to check system correctness and maximize the efficiency of the overall verification process.

In order to achieve *completeness* (i.e. a certainty that the system does not violate its specification) of the process of verification some formal techniques and tools can be used. They are adequate for detecting errors in highly concurrent and complex designs where traditional ways mentioned before are not sufficient.

Today's highly competitive market of consumer electronics is very sensitive to the time it takes to introduce a new product (the so-called *time to market*). This has driven the demand for fast, efficient and cost-effective methods of verification of hardware systems. Simulation-based approaches suffer from the fact that software simulation of inherently parallel hardware is extremely slow when compared to the speed of real hardware. The gap between the speed of simulation and the speed of real hardware widens with the increasing complexity of the hardware design. For complex circuits a simulation of several thousand

clock cycles may take hours or even days.

However, only recently with the advent of sophisticated programming languages for hardware verification, such as SystemVerilog [14], and standardized verification methodologies (e.g. OVM [13], UVM [8]), have there appeared products that target the bottleneck of simulation by acceleration, for instance Schwarztrauber's SEmulation [18] or Mentor Graphics' Veloce [5] technology. These are mainly proprietary solutions.

The aim of this work is to design and implement an open framework that exploits the inherent parallelism of hardware designs to accelerate functional verification of these designs using the FPGA technology, while retaining the possibility to run the same verification runs in the user-friendly debugging environment of a simulator.

We use the NetCOPE platform for handling data transfers and communication through *direct memory access* (DMA) channels between the software and the hardware part of the verification environment. Two interface protocols (FrameLink and MI32) are introduced because they allow to build a communication layer between the NetCOPE platform and the verification core placed in the FPGA. In order to communicate with and transfer data to and from the verification core using the NetCOPE-provided libsze2 library functions, the SystemVerilog testbench uses the *direct programming interface* (DPI), which allows to call those functions written in the C language directly from a SystemVerilog testbench.

The text is divided into several chapters. Chapter 2 introduces the history of techniques and tools for hardware design and verification. This is followed by theoretical principles of widely used verification approaches: simulation and testing, functional verification, formal analysis and verification. Chapter 3 describes the SystemVerilog language, its features and steps to build effective and reusable verification environments. Chapter 4 discusses available related work in the field of hardware acceleration of simulation and functional verification. Chapter 5 analyses the requirements on a framework for acceleration of functional verification. Chapter 6 describes in detail the design of the framework and Chapter 7 describes the implementation. Chapter 8 demonstrates the use of the framework when debugging an erroneous component and Chapter 9 presents the results of the experiments on a prototype implementation. Finally, Chapter 10 summarizes the work and outlines its possible further development.

Chapter 2

Verification Techniques

The goal of hardware design is to create a device with a particular functionality. The task of a verification engineer is to make sure that the device implements the functionality properly, which can be assessed by verification that checks whether given system (a real system or a model) satisfies given correctness specification [16]. This chapter describes in detail three different approaches to verification of hardware designs.

2.1 History of Design Verification Methods

Over the years, as technology progressed and the complexity of hardware designs increased, new verification methods were needed and developed. During the last 40 years many advanced and effective techniques and tools for verification appeared (the history overview is taken from [10]).

In the 70's, hardware was designed by drawing schematics on the paper, later electronically, but verification was mainly performed by detailed review of the schematics. During the 80's, simulation tools became popular, but they were predominantly proprietary. Testbenches were hard to build with non-standardized tools, and verification relied on manual assessment of simulation results. In 1987, VHDL became a standard and *hardware description languages* (HDLs) in general were accepted as means of design and verification.

Static verification tools also started to emerge around this time in order to help in static analysis determining whether designs conform to design rules. As the density of designs increased, the use of more intelligent testbenches became a common practice. Self-checking testbenches with directed testing were then followed by testbenches using *constrained-random stimulus generation* and finally *transaction-based* testbenches. The implementation of advanced testbenches along with the introduction of better tools (coverage checkers, faster simulators) increased rapidly the popularity of simulation. However, as designs became even more complex, HDLs of that time were not sufficient for effective verification, since only very restricted functions and data types could be defined.

To overcome these deficiencies, *hardware verification languages* (HVLs) were introduced. These HVLs were tailored for simulation and integration with HDLs. HVLs introduced the concept of *functional coverage* and helped in tracking the progress of verification. Evolution of verification methodologies and languages led towards standardization of the SystemVerilog language (its current standard being IEEE 1800-2009 [14]).

Assertion-based verification is a concept originally introduced in the scope of formal (static) verification. This approach uses a simple language based on a temporal logic (such

as *linear temporal logic* (LTL) or *computation tree logic* (CTL)) to describe specifications of parts of a system in the form of a set of temporal formulae. In the case of formal verification tools, the state space of the system is then exhaustively searched to check whether all specified temporal formulae are valid in all accessible states of the verified system. In case a state that invalidates some formula is found a *counter-example* (also called a *witness*) is given in the form of a waveform of a sequence of input and internal signals that leads to the failing state. In case of dynamic verification tools, in each clock cycle of the simulation all formulae are checked for validity and in case of a failure the simulation is stopped with an error message.

2.2 Simulation and Testing

Simulation and testing are called *bug hunting* methods as their main purpose is to find as many bugs as possible. These methods are only able to uncover faults, they cannot guarantee their absence, i.e. they are not complete.

One of the main strengths of a simulation is that software simulators provide a perfect debugging environment. All the signals in the design are readily accessible. Once an error has been reproduced in this environment the process of debugging can be performed extremely efficiently. For this it is crucial that the same failing scenarios are reproducible in the simulator.

The disadvantage of simulation is its low performance which directly depends on the complexity of the simulated design. Practically it can take days or potentially weeks to simulate a large design because simulation is a computationally intensive task. This is a reason why a detailed and thorough software simulation can be often performed only on small portions of the design.

2.3 Formal Analysis and Verification

Formal methods are based on formal mathematical roots. Unlike other approaches, formal verification is capable of proving the correctness of a given system according to a given specification and not just disprove a property according to some observed behaviour which is the case of simulation and testing. As the task of formal verification is in general undecidable, a formal verification method may not guarantee termination, and even when termination is guaranteed the complexity of the verification task may call for some method of smart state space exploration. Some methods deal with the issue of termination using abstraction through upper-approximation, which may in turn introduce false alarms. Alternatively, a method is allowed to stop with a "don't know" answer or it may become not fully automated (some human help is required). Even if a full formal verification of a system fails, it may still be useful as it can find some errors in the meantime.

There are several distinct verification techniques used in this formal field. The most important and best known are:

- **Model Checking** is an algorithmic approach of checking whether given system satisfies a given property through a systematic search of the state space of the system. It is a technique for automatic and exhaustive verification of software and reactive systems which can be modelled by a finite automaton (or a variant of this general representation). Properties are classically specified using temporal logics such as CTL, CTL*, and LTL. It is a successful method frequently used to uncover well-hidden

bugs. Typically the goal of this technique is to answer the question: "Does model M satisfies property P?". A positive reply guarantees the underlying property for all behaviours of the model (at the level of a mathematical proof). A negative reply is usually accompanied by a counter-example: a particular run of the system which leads to a violation of the desired property. The main obstacle encountered by model checking algorithms is the *state space explosion* problem. A system easily handled by a simulator, a compiler or an interpreter may be unable to be verified in a reasonable time by a model checker. In practice it also requires a good expertise from the user to deal with formulation of properties to be verified.

- **Static Analysis** is an approach with high level of automation which tries to avoid execution of the system being examined by analyzing its source code instead. It also gathers some typically approximate information about the system from the source code. There are different forms of static analysis: type analysis, bug pattern searching, equational dataflow analysis, constrained-based analysis, or abstract interpretation. Static analysis is not exclusively intended for checking correctness of systems only, it is used also for optimization, code generation, etc. The main advantages are that it can handle very large systems and does not need a model of the environment of the system. However, it can produce many false alarms and various analyses are specialized just for a certain specific task.

- **Theorem Proving** is a deductive verification method often similar to the classical mathematical way of proving theorems starting with axioms and inferring further facts using rules of correct inference. This approach is very general but semi-automated as it often requires a significant manual effort of users.

In conclusion there is no ideal formal technique that would allow fully automated proofs for all types of systems. The choice of the best suited approach strongly depends on the actual verification problem. In practice only critical system parts are verified formally.

2.4 Functional Verification

Functional verification is based on simulation but uses more sophisticated testbenches with additional features like self-checking mechanisms, constrained-random stimulus generation or coverage-driven verification to achieve controllability and observability of the verification progress. This approach is described in more detail in Chapter 3.

Chapter 3

SystemVerilog Functional Verification

SystemVerilog is a complex programming language for hardware description and verification. While created as the next generation of the Verilog language, it has adopted features from many other programming languages with great impact on its simulation and verification capabilities. SystemVerilog offers a lot of techniques to increase efficiency of the verification process. The description of some of these techniques follows:

- **Object-oriented programming (OOP).** This approach allows easier design of large systems with support of common design patterns or reusable components. Testbenches are more modular and thus easier to develop and debug. The mechanisms of encapsulation, inheritance and polymorphism support the reuse of verification components, which leads to an increase in productivity. There are several orders of reusability. The first order is when the same verification environment is used across multiple testcases on the same project. The second order occurs when some verification components are used several times in the same project environment. The third order is when some verification components are used across different verification environments for different designs.

- **Constrained-random stimulus generation.** For checking full functionality of a larger design it becomes more difficult to create a complete set of stimuli. A suitable solution is to create testcases automatically using constrained-random stimulus generation to target corner cases and stress conditions. Test scenarios are restricted to be valid using constraints. Constraints can also be used to guide tests to interesting DUT states. Due to randomly generated inputs, it is necessary to use coverage mechanisms to explore the DUT's state space evenly.

- **Assertion-based verification (ABV).** This is a technique used to formally express the intended design behaviour, internal synchronization, and expected operations, using assertions (i.e. properties that must hold at all times). Assertions can be expressed at many levels of the device including internal and external interfaces (to detect critical protocol violations), clock-domain crossings and state machines. Assertions create monitors at critical points of the design without having to create separate testbenches where these points would be externally visible. ABV also guides the verification task and quickens the verification because it provides feedback at the internal level of the device as it is possible to locate the cause of a problem faster than from the output of

a simulation. Two examples of assertion languages are PSL (Property Specification Language) and SVA (SystemVerilog Assertions).

- **Functional coverage.** This is a technique that provides detailed feedback about which features of the design have been appropriately checked. It measures testcases and interesting conditions to capture the progress and productivity of the verification process. To express legal input stimulus, directed tests or constrained-random stimulus generation tests are used. Some techniques even enable to manage the generation of appropriate scenarios in order to converge to the 100 % coverage. This process can be fully automated by an intelligent program that controls coverage results and chooses parameters or a pseudo-random number generator seed according to achieved coverage (coverage-driven verification). Note that this technique cannot ensure completeness of the functional verification.

- **Code coverage.** Code coverage provides another metric to assess verification completeness by measuring the proportion of structural code constructs exercised during simulation. This includes several metrics such as line coverage (how many lines of code have been executed), path coverage (which paths through the code have been executed), toggle coverage (which single-bit variables have switched from 0 to 1 or 1 to 0), expression coverage (which expressions have been executed), and FSM coverage (which states and transitions in a *finite state machine* have been visited). No extra HDL code needs to be written because the code coverage tool included in many simulators instruments the design automatically by analyzing the source code and adding hidden code to gather statistics. It is desirable to reach the highest degree of coverage as untested code may conceal errors.

- **Cooperation with other programming languages.** *Direct programming interface* (DPI) allows SystemVerilog code to call functions in other programming languages as if they were native SystemVerilog functions. Data can be passed between the two domains through function arguments and results. Interoperable environments and components may be used to reduce the effort required to verify a complete product in case some parts of the product are already prepared in other programming languages.

3.1 Verification Process

Verification is a challenging task, therefore a lot of effort should go into specifying when a DUT is correct and fulfils the intended function according to the specification. Inspired by [9] we introduce the main steps of a verification process (Figure 3.1):

3.1.1 Specification and Requirements

In order to check a new implementation for functional correctness we need a reference description, either a specification or a previous *golden model* reference implementation which represents the intention of the design. In many cases the specification is given on a higher level of abstraction so it does not capture the detailed behaviour of the design. The requirements for the appropriate verification process are built according to the specification of the functionality of the device. The requirements should be defined as soon as possible

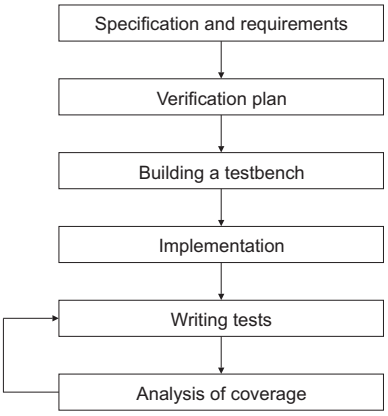


Figure 3.1: Steps of the verification process.

in the project life cycle, ideally while the architectural design is in progress. It is recommended that the requirements are identified by all members of the project team: hardware and software developers and verification engineers. A few examples of requirements are: definition of legal input and output patterns according to protocols on interfaces, description of incorrect behaviour of the device, identification of potential corner cases, supported configurations, errors from which the design can or cannot recover and how the system responses and behaves in such cases.

3.1.2 Verification Plan

Verification plan helps verification engineers to understand how verification should be done. It is closely tied to the hardware specification and requirements and contains a description of which features need to be exercised and techniques to be used to achieve specific goals so that the progress could be easily measured.

- *Stimulus generation plan* chooses the character of input sequences:

  a) **Directed tests** — each test contains directed stimulus sequences which are targeted at a very specific set of design elements. The DUT is stimulated with these sequences and the resulting log files and waveforms are then manually reviewed to make sure that the design behaves according to the specification. Directed tests typically find errors that are expected to be in the design and cover the state space very slowly. Creating complex stimulus set is a very time-consuming and challenging task because this approach requires a verification engineer to write many tests in order to achieve high coverage.

  b) **Constrained-random stimulus generation tests** — as designs become more complex it is impossible to use only directed tests. A more efficient way to verify complex designs thoroughly is with constrained-random stimulus generation. The widest possible range of stimuli can help find bugs which were not anticipated. Random tests explore the space much faster than directed tests, reduce the number of required tests, and increase productivity and quality of the verification process.

- *Checker plan* uses mechanisms for predicting the expected response and for comparing the observed response (typically from external outputs) against the expected one. These checks should be intelligent and independent of testcases. The following list introduces several means of predicting expected responses.

  a) **Assertions** — these are used to verify the response of the device based on internal signals. Any detected discrepancy in the observed behaviour results in an error which is reported near to the origin of the functional defect. Assertions work well for verifying local signal relationships; they can detect errors in handshaking, state transitions and protocol rules. On the other hand, they are not well suited for detecting data transformations, computation and ordering errors. Despite the effectiveness of assertions, they are limited to the types of properties that can be expressed only using expressions in a clock temporal logic.

  b) **Scoreboarding** — a scoreboard is used to dynamically predict the response of the device. Stimulus applied to the device is also passed to the transfer function which performs all transformations on the stimulus to produce the form of the final response. Modified stimulus is inserted to a data structure. The observed response from the DUT in the form of the transaction is forwarded to the comparison function which verifies whether it is the expected response or not (Figure 3.2).

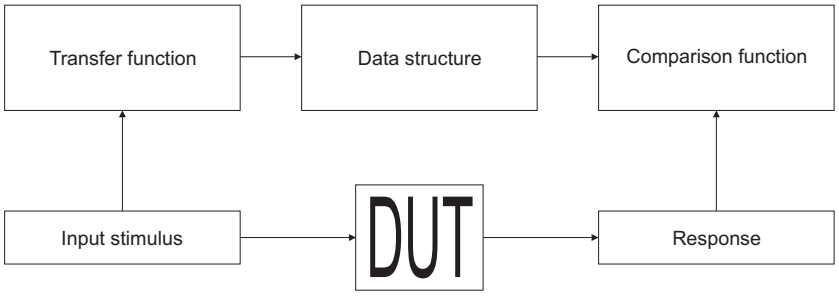


Figure 3.2: Scoreboarding.

  The transfer function may be implemented using a reference or a golden model, even e.g. in the C language (and integrated into the testbench through the DPI). The data structure stores the expected response until it can be compared against the observed output. Scoreboarding works well for verifying the end-to-end response of a design and the integrity of the output data. It can efficiently detect errors in data computations, transformations and ordering. It can also easily identify missing or spurious data.

  c) **Reference model** — like a scoreboard, it is used to dynamically predict the response of the device. The stimulus applied to the design is also passed to the reference model. The output of the reference model is compared against the observed response (Figure 3.3). The reference model mechanism has the same capabilities and challenges as the use of scoreboards. Unlike a scoreboard, the comparison function works directly from the output of the reference model so

that it has to produce outputs in the same order as the design itself. However, there is no need to produce the output with the same latency or cycle accuracy. The comparison of the observed response is performed at the transaction-level, not at cycle-by-cycle level. The use of reference models depends heavily on their availability. If they are not available, scoreboard techniques are more efficient to be used. Reference models are often written in the C language and therefore easily exploitable by a SystemVerilog testbench using the mechanism of the DPI.

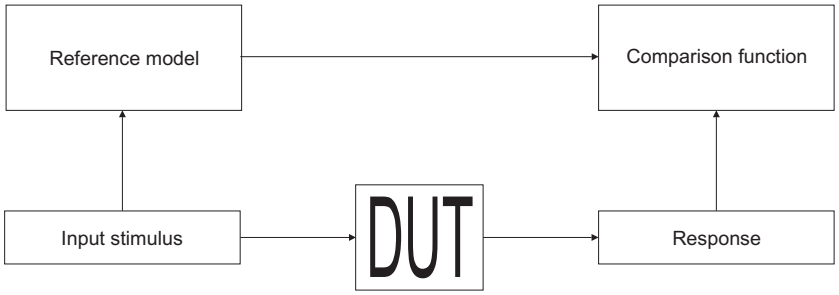


Figure 3.3: Reference model.

d) **Accuracy** — the simplest comparison function compares the observed output of the design with the predicted output on a cycle-by-cycle basis. This approach requires the response to be accurately predicted down to the cycle level.

e) **Offline checking** — used to predict the response of the design before or after a simulation of the design. In pre-simulation prediction, the offline checker produces a description of the expected response, which is dynamically verified against the observed response during simulation (Figure 3.4). Some utilities can perform post-simulation comparison (Figure 3.5). In both cases the response can be checked at cycle-by-cycle or transaction level with reordering. Offline checkers work well for verifying the end-to-end response of a design and the integrity of the output data based on executable system-level specifications or mathematical models. Although offline checkers are usually used with a reference model, they can be used also with scoreboard techniques implemented as a separate offline program.

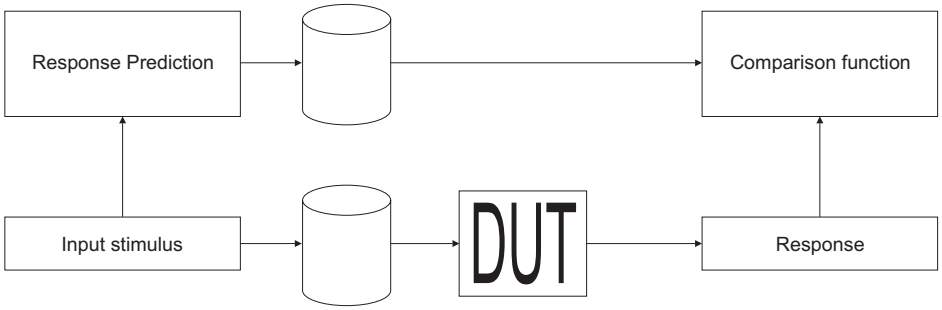


Figure 3.4: Pre-simulation offline checking.

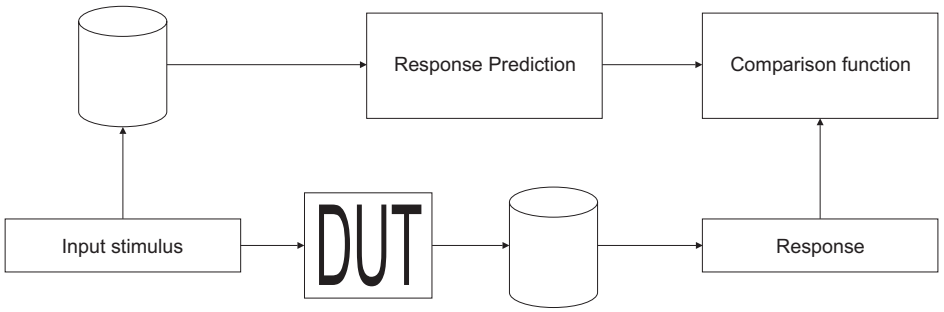


Figure 3.5: Post-simulation offline checking.

3.1.3 Building a Testbench

The verification environment of the DUT is developed in this step. The purpose of the testbench is to determine the correctness of the DUT. This is accomplished in general by:

1. generating stimuli,
2. applying stimuli to the DUT,
3. capturing the response,
4. checking correctness of the response,
5. measuring progress against the overall verification goals.

The continuous growth in the complexity of hardware designs requires a modern, systematic and automated approach to creating testbenches. One of the big challenges in developing testbenches is to make effective use of the object-oriented programming paradigm. When used properly, these features can greatly enhance the reusability of testbench components.

Functional verification testbenches are usually complex so it is highly desirable to divide the code into smaller pieces that can be developed separately. Some general components are present in all verification environments nowadays (Figure 3.6):

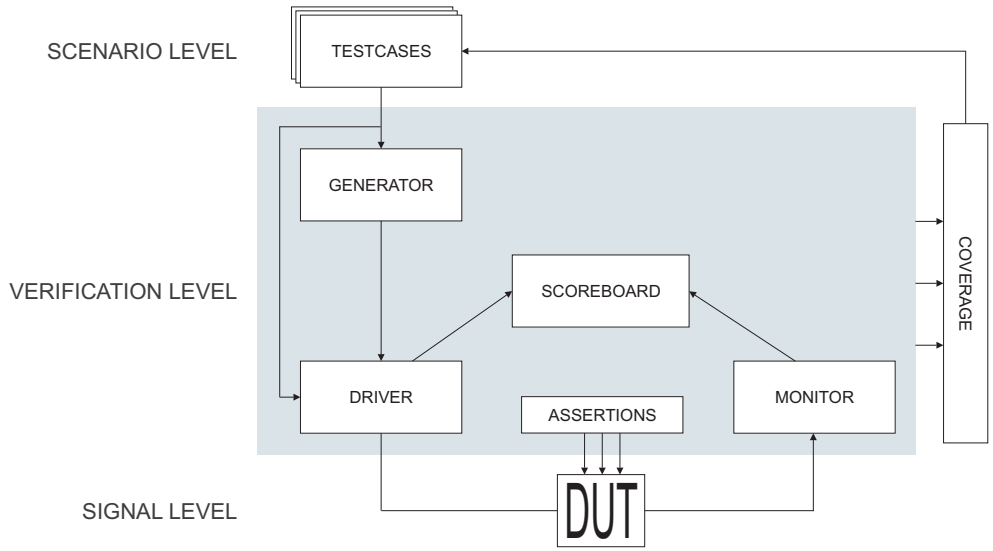


Figure 3.6: A verification environment.

- Scenario level:
  - Testcases are situated on top of the verification environment. They are implemented as a combination of additional constraints for generators, new random scenario definitions, synchronization mechanisms between transactions, or directed stimuli.
- Verification level:
  - The generator produces constrained random stimuli and passes them to the driver.
  - The driver receives high-level directed transactions or transactions from the generator, breaks them down into individual signal changes, supplies them on input interfaces of the DUT, and sends the copy to the scoreboard. The driver may inject errors and add delay parts.
  - The monitor drives DUT's output interfaces, observes signal transitions, groups them together into high-level transactions and passes them to the scoreboard.
  - The scoreboard compares transactions received from the driver and the monitor and in the case they do not match reports an error.
  - Assertions check the validity of protocols on interfaces of the DUT. In the case of violation they directly report the failing assertion.
- Signal level:
  - Verified hardware design (DUT) is connected to verification level components through a set of signals. This layer provides pin name abstraction to enable the whole verification environment to be used unmodified with different implementations of the same DUT.

3.1.4 Implementation

Verification environments and testcases should access the design only through the verification level and never access the signal level unless absolutely necessary.

3.1.5 Writing Tests

After the testbench is applied to the DUT it is the time for validating the DUT. By analyzing coverage reports, new tests are written to cover holes in coverage using two different approaches. The first method captures a new run of the simulation with a different seed of the generator or tailors constraints on input stimuli (creating many unique input sequences with constrained-random stimulus generation). The other approach represents creating directed tests. This iterative process can be seen in Figure 3.7.

3.1.6 Analysis of Coverage

Coverage tools gather information during a simulation and post-process them in order to produce a coverage report. After analysing both functional and code coverage reports, new tests are written to reach uncovered areas of the design until a sufficient level of coverage is achieved.

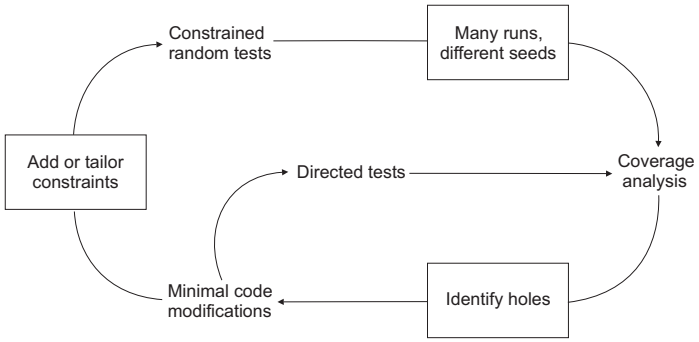


Figure 3.7: Analysis of coverage.

3.2 Verification Methodologies

The success of verification does not depend only on the verification language. The target is to create an applicable verification methodology that uses common approach and creates highly interoperable verification environments and components. An effective methodology must address the main verification challenges: productivity and quality. Coverage-driven verification coupled with constrained-random stimulus generation are currently becoming the main principles of verification methodologies. A description of three most popular verification methodologies follows.

- **Verification Methodology Manual (VMM)** [9] — was co-authored by verification experts from ARM and Synopsys. VMM's techniques were originally developed for use with the OpenVera language and were extended in 2005 for SystemVerilog. VMM has been successfully used to verify a wide range of hardware designs, from networking devices to processors. It describes how to build scalable, predictable and reusable verification environments. Very useful is the set of base classes for data, environment and utilities for managing log files and interprocess communication. VMM is focused on the principles of coverage-driven verification and follows the layered testbench architecture to take full advantage of the automation when each layer provides a set of services to the upper layers while abstracting them from lower-level details.
- **Open Verification Methodology (OVM)** [13] — this methodology is the result of a joint development between Cadence and Mentor Graphics to facilitate true SystemVerilog interoperability with a standard library and a proven methodology. The class library provides building blocks which are helpful for fast development of well-constructed and reusable verification components and test environments in SystemVerilog. The OVM class library contains:
  - component classes for building testbench components,
  - reporting classes for logging,
  - factory for object substitution,
  - synchronization classes for managing concurrent processes,
  - sequencer and sequence classes for generating realistic stimuli.
- **Universal Verification Methodology (UVM)** [8] — is a state of the art methodology that extends OVM.

Chapter 4

Functional Verification Acceleration Techniques

Nowadays we can observe significant effort of many researchers and companies to increase efficiency and speed of simulation and verification techniques. We chose a few of them that helped us comprehend the issue of acceleration and also to formulate our own acceleration method.

4.1 FPGA Prototyping

Hardware-assisted verification environments often make use of FPGAs to prototype the ASIC in order to provide a faster alternative to simulation and allow software development to proceed in parallel with hardware design [17, 15]. The main advantages are:

- **Faster simulation speed:** much closer to real-time operation enabling much more data to be processed and if controlled correctly perhaps more corner cases to be stimulated.
- **Realistic system environment:** where possible the device can be connected to known system components and external interfaces.
- **Software development platform:** the improved performance allows concurrent development of software with the evolving hardware.

FPGA prototyping does not replace simulation-based verification, but rather it is a technique to improve the overall effectiveness of the verification. The main reasons for this are that FPGA is not the final product and that the FPGA prototyping environment lacks many essential features provided by simulation-based verification (different physical implementation, unimplemented or modified modules, poor debugging environment, etc.).

4.2 Substituting Testbench Components with their Simulation Models

Freitas [12] describes a highly effective way to use the SystemVerilog DPI to integrate software tools written in the C language (an *instruction set simulator* (ISS) and a software debugger) into a logic simulation of the Hyperstone S5 flash memory controller. The benefits

of this approach include testing and integration of code earlier in the design cycle and the ability to more easily and quickly reproduce problems in the simulation.

A substantial part of the system was implemented in a firmware, which made hardware/software co-design and co-verification very advantageous because of poor performance of the simulation. The acceleration was accomplished by replacing the gate-level representation of the microprocessor with its ISS written in the C language. Because the ISS comprises all the memories of the system, processor cycles used to fetch the code and read and write static variables were no longer simulated by the logic simulator. This drastically reduced the simulation time. To integrate the ISS with logic simulation the SystemVerilog DPI was used.

4.3 Simulation Acceleration Using an Emulation Board

Schwarztrauber [18] proposes the use of the SEmulator emulation platform as a hardware accelerator for RTL-simulation. This work connects simulation with emulation using FPGA prototyping. Design blocks can be easily moved from the simulator into the hardware without leaving the simulation environment and without the need to recompile the complete design for every minor modification, thus shortening the development time. SEmulator can use a FPGA or standard emulation hardware like ARM's Microcontroller Prototyping System, Altera's Automotive Platform (PARIS) or even customer specific emulation hardware as a hardware accelerator for RTL-simulation of complex designs, speeding up simulation by a factor of 100 and more. Of course this factor depends on the complexity of the design blocks that have been moved to hardware. In the simulation process, an additional step has to be made: the designer has to specify the design blocks that are intended to remain in software and the ones that are supposed to be moved into hardware. Signals cannot be observed during hardware acceleration.

4.4 Synthesis of Testbenches

The work of Daw *et al* [11] first introduces co-simulation approaches to increase the overall simulation speed. A behavioural testbench runs in a software simulator and the RTL design is executed in a reconfigurable hardware platform which may be implemented as a set of general-purpose processors or FPGAs. These need to communicate with each other in order to maintain synchronization. The consequence of such frequent communication is a reduction of the potential speed at which the system may operate. Because of this limitation, co-simulation is typically only 3 to 10 times faster than a software simulation.

This technique is later improved by mapping the testbench to the reconfigurable platform as well. The work describes a set of compilation transformations that convert behavioural constructs of testbenches into RTL constructs that can be directly mapped onto a FPGA. Such transformations are provided by introducing the concepts of a behavioural clock and a time advance finite state machine that determines simulation time and sequences concurrent computing blocks in the DUT and the testbench. The entire design and testbench can now run on a hardware platform achieving much better simulation performance. The larger the design is, the better the performance gain (more than 1,000 times of the software simulation speed). A benefit of this is that designers and verification engineers may achieve such a gain without any changes to their current verification methodology.

4.5 Veloce

One of the commercial products dealing with acceleration of simulation and high performance in-circuit emulation is Mentor Graphics' Veloce technology [5]. The Veloce product line supports OVM and ABV. It allows acceleration of transaction-based test runs by 100s to 1,000s of times. The main advantages of this technology are faster runtime performance, faster design compilation, comprehensive simulation-like debugging environment, scalable architecture or full system integration using real-world data. It achieves fast compilation times thanks to a unique network for interconnecting computational resources. Veloce even supports the capability to independently compile the logic part of the design from the interconnection part and also to model an asynchronous behaviour of a device. Communication between the software testbench and the DUT in hardware is accomplished using a testbench interfacing technology called TBX, which is based upon the SCE-MI 2.0 standard [7]. This standard allows high-level implicit management of communication using procedural calls.

Chapter 5

Analysis

As stated in Chapter 1, the duration of a verification process continues to grow as a consequence of increasing complexity of hardware systems, therefore hardware-assisted acceleration is highly desirable.

Our goal is to develop an acceleration framework that would enable writing high-level functional verification testbenches in SystemVerilog which could be easily accelerated without the use of expensive specialized emulation boards as needed in the approaches mentioned in Chapter 4. This is possible because the generic nature of verification methodologies and transaction-based communication among their subcomponents make it possible to transparently move these subcomponents to a specialized hardware (FPGA), while maintaining the same level of readability to verification engineers.

One of the features of the framework should be to allow the user to run either the non-accelerated or the accelerated version of the same testbench, even with the same time behaviour.

5.1 Non-accelerated Version

The non-accelerated version (also called the software version) of the framework presents a similar approach that is commonly used in verification methodology libraries. This version is useful in the initial phase of the verification process when debugging basic system functions. In this phase it is desirable to have a quick access to the values of all signals of the design and to monitor the verification process in a simulator.

5.2 Accelerated Version

The accelerated version of the framework moves the verified component to a verification environment in the FPGA. As gate-level simulation takes the biggest portion of verification time, this approach may yield a significant acceleration of the overall process. Behavioural parts of the testbench, such as planning of test sequences, generation of constrained-random stimuli, monitoring coverage reports, and scoreboarding, remain in the software simulator. If an error occurs the user can start the failing verification scenario with the same time behaviour in the non-accelerated version for comfortable debugging.

5.3 Reproducing Failing Scenarios

In case a failing scenario is detected in the accelerated version of the framework, in order to ensure that the same failure occurs also in the non-accelerated version, it must be guaranteed that the sequence of values of signals is the same. This makes it easy to debug the failure in a simulator.

Chapter 6

Design of a Verification Framework

The components of the verification environment differ according to the selected version of the framework as illustrated in Figure 6.1.

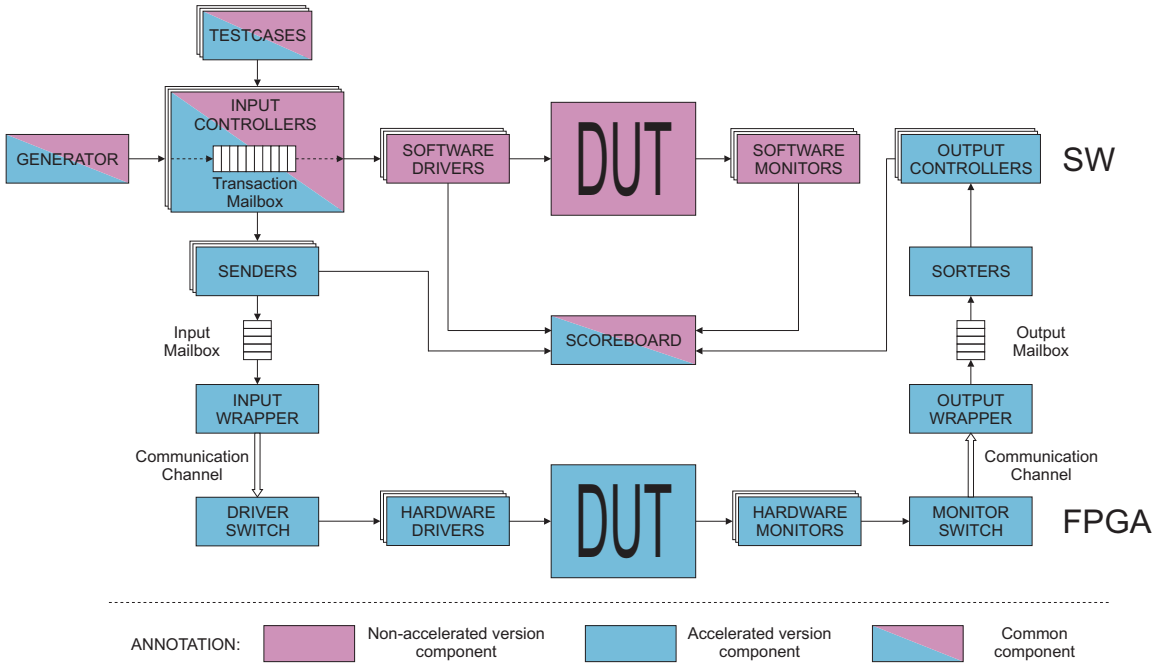


Figure 6.1: The verification environment of the framework.

A detailed description of both framework versions follows. The remaining part of this section deals with implementation details.

6.1 Non-accelerated Version

The components of the non-accelerated version are written in the SystemVerilog language. Knowledge of verification methodologies and object-oriented nature of SystemVerilog enable to implement interoperable, reusable and scalable verification environments. We can easily assemble packages of basic verification components (classes) as well as packages of user-defined components typically inherited from basic classes. Basic components of the

non-accelerated version of the framework are illustrated in Figure 6.2 and their detailed description follows.

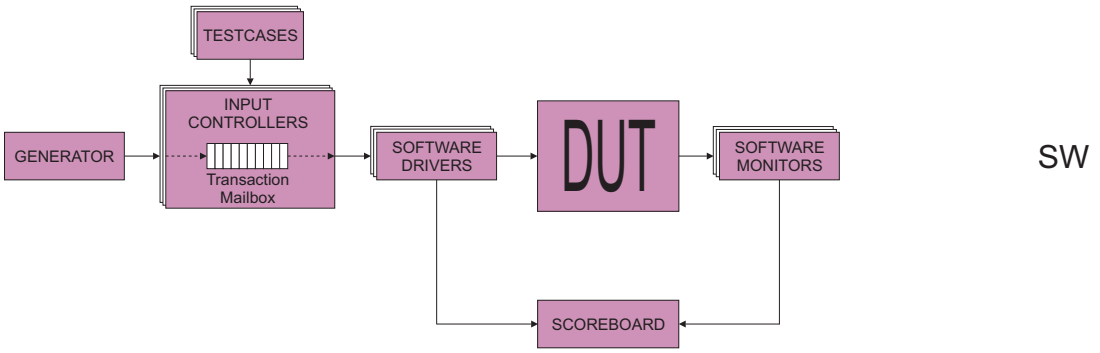


Figure 6.2: The non-accelerated version of the verification environment.

Design Under Test (DUT) — the verified hardware system which is connected to the software verification environment through a set of interface signals created in SystemVerilog. During the software version of verification the DUT runs in a simulator.

Testcases — are written by a user, who builds verification environment either from available components (defined in SystemVerilog packages, or methodology libraries) or her own components. The Testcase is the reserved place for setting generics and parameters like transaction count, delays between or inside transactions or the version of the framework to run. The user creates a test sequence, which contains instructions for the Input Controller to manage constrained-random stimulus generation, sending of directed stimuli or synchronization between transactions. We support two types of transactions: data transactions, which are either randomly generated or direct, and control transactions (`start, wait, waitforever, delay, stop`). They are described in detail in Section 7.6.

Input Controllers — interpret instructions from testcases. The class hierarchy defines basic Input Controllers to manage randomly generated or direct transactions. If random stimuli are required, it is necessary to call the software Generator. For the following processing of transactions, the Input Controller hands over the control to the Software Driver. It is recommended to create an independent Input Controller for every interface protocol through inheritance from basic classes.

Generator — produces constrained random stimuli, which are typically random data and random delays between and inside transactions in the range specified in the Testcase. SystemVerilog provides a set of functions that can be used for randomization.

Software Drivers — receive instructions from assigned Input Controllers in the form of task calls and also transactions through the Transaction Mailbox. The driver breaks data transactions down into individual signal changes and supplies them on the assigned input interface of the simulated DUT. The copy of the data transaction is sent to the Scoreboard. The driver may also inject errors or add delay parts.

Software Monitors — drive simulated DUT's output interfaces, observe signal transitions, group them together into high-level data transactions and pass them to the Scoreboard.

Scoreboard — compares expected transactions (received directly from a Software Driver or modified according to the evaluation function of the DUT) with transactions received from a Software Monitor. If they do not match, the Scoreboard reports an error. The Scoreboard contains special functions for preprocessing (data transformation before sending a transaction to the DUT, e.g. to meet special requirements of the input interface protocol) and postprocessing (data transformation before sending the transaction to comparison function in the Scoreboard to catch data modification in the DUT).

6.2 Accelerated Version

The components of the accelerated version are divided into the software and the hardware parts of the verification environment. Software components are written in the SystemVerilog language and hardware components in the VHDL language. Basic components of the accelerated version of the framework are illustrated in Figure 6.3 and their detailed description follows.

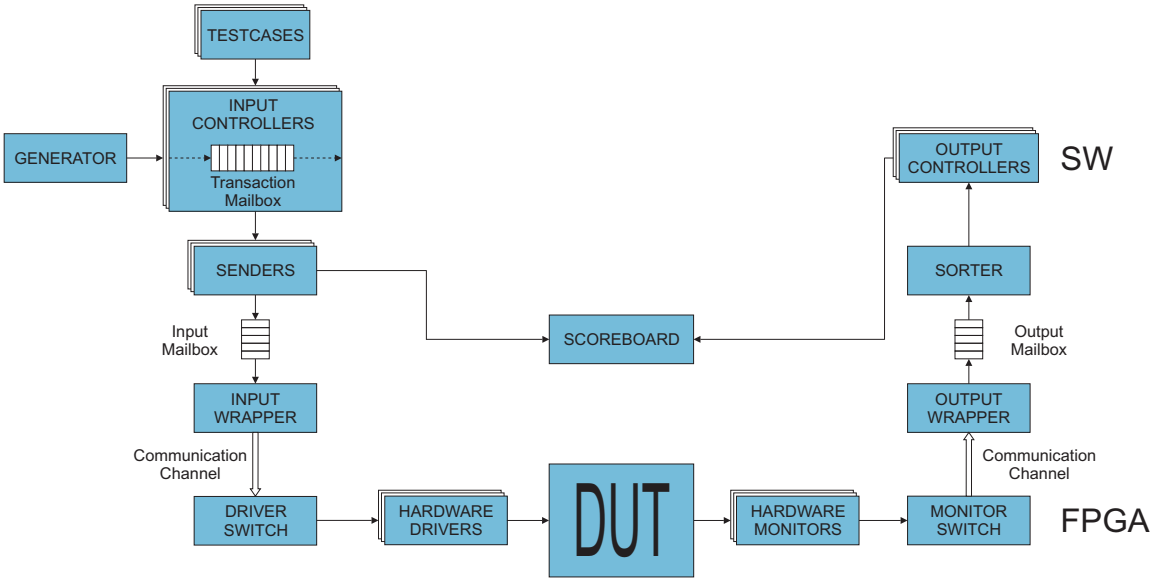


Figure 6.3: The accelerated version of the verification environment.

Design Under Test (DUT) — the verified hardware system is synthesized and placed in the reconfigurable hardware platform (FPGA). This can fully exploit the inherent parallelism of hardware.

Testcases — are made according to the same principles as in the non-accelerated version. But in addition, the user builds verification environment not only from software components but also connects hardware components. These are defined in special hardware packages.

Input Controllers — interpret instructions from testcases similarly as in the non-accelerated version, but for the following processing of transactions an Input Controller hands over the control to a Sender. It is also recommended to create an independent Input Controller for every interface protocol through inheritance from basic classes.

Generator — produces constrained random stimuli.

Senders — receive instructions from assigned Input Controllers in the form of task calls and data transactions through the Transaction Mailbox. The copy of a received data transaction is sent to the Scoreboard. According to the instruction, the Sender builds data and control transactions for a specific component in the hardware part of the verification environment. In order to deliver the transaction to the correct Hardware Driver, the Sender provides each transaction with transfer protocol headers. Transactions created by all Senders are buffered in the Input Mailbox.

Input Wrapper — opens a communication channel with the hardware platform at the beginning of the verification run and closes the channel after all prepared data are sent.

Driver Switch — routes received transactions from the software to the proper Hardware Driver according to the fields in transfer protocol headers.

Hardware Drivers — perform similar function as their non-accelerated counterparts. The Hardware Driver receives transactions from the Driver Switch and checks transfer protocol headers. Depending on the information in headers the Driver recognises the nature of the transaction: whether it is data or control. In the case of a data transaction the Driver parses the data and propagates them as signal changes to the proper input interface of the DUT. If a control transaction occurs, the Driver performs a specific task depending on the type of the control transaction. Moreover a single driver can manage several interfaces connected to the DUT.

Hardware Monitors — drive DUT's output interfaces, observe signal transitions and group them together into high-level data transactions. For correct delivery to the proper software Output Controller they also add transfer protocol headers to every data transaction.

Monitor Switch — collects transactions from Hardware Monitors and prepares them for transmission through the communication channel to the software part.

Output Wrapper — retrieves transactions from the hardware part using the communication channel. After transactions are correctly received, the Output Wrapper buffers them in the Output Mailbox.

Sorter – takes transactions from the Output Mailbox and routes them to the proper Output Controller according to the fields in transfer protocol headers.

Output Controllers — receive transactions from the Sorter and extract transfer protocol headers. According to the information in headers, an Output Controller classifies transactions to data and control. Data transactions are sent in the correct format to the Scoreboard and control transactions are appropriately interpreted.

Scoreboard — in the accelerated version compares expected transactions (received directly from the Sorter or modified according to the evaluation function of the DUT) with transactions received from an Output Controller. If they do not match, the Scoreboard reports an error. Again there are special functions for preprocessing and postprocessing of data transactions.

6.3 Transfer Protocol Stack

To maintain transaction-based approach and integrity in our design of the verification process it is necessary to design special purpose transfer protocol stack for transmission of data between software and hardware part of the verification environment.

The stack contains three abstract layers as illustrated in Figure 6.4. They specify how the data should be formatted, addressed, transmitted, routed and received at the destination.

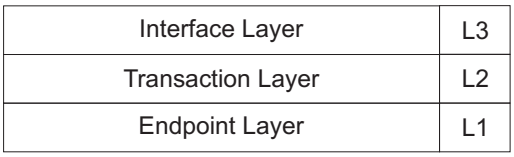


Figure 6.4: Layered stack for communication between the hardware and the software part of the verification environment.

Endpoint Layer — identifies the destination unit for every transmitted transaction. In the case of communication from software to hardware it identifies typically one of Hardware Drivers and in the case of communication from hardware to software it identifies one of Output Controllers.

Transaction Layer — defines the type of transmitted information. At this level it is possible to make a difference between data and control transactions.

Interface Layer — identifies interface of the verified hardware system either physical in hardware or virtual in software.

Chapter 7

Implementation Details

This chapter describes the NetCOPE platform for development of hardware accelerated applications. First, communication between the hardware and the software is described, followed by the specification of two interface protocols: FrameLink and MI32. Then we specify the DPI interface for communication between the SystemVerilog and the C programming languages. Further, the implementation of the protocols of the designed communication stack is explained. The remaining part of the chapter is dedicated to the issue of maintaining reproducibility of verification on the level of implementation.

7.1 The NetCOPE Platform

We decided to choose the NetCOPE platform for handling communication and data transfers between the software and the hardware part of the verification environment in the accelerated version of the framework. The NetCOPE platform was primarily designed for fast development of network applications on the Liberouter project [3], but it can be properly used as an acceleration platform too. The platform provides an abstract layer between an operating system (software drivers and libraries) and a FPGA (especially DMA transfers between the acceleration card and the computer's RAM memory through the PCI Express (PCIe) bus). Thanks to the unified interface of hardware application we can achieve total abstraction from a physical card. Nowadays the NetCOPE platform provides an abstract layer over the whole family of COMBO cards [2].

From the point of view of acceleration of verification we do not need special network interfaces provided by the NetCOPE platform (two input and output buffers connected to 10 Gb Ethernet interfaces on the chosen *COMBOv2 LXT155-10G2* acceleration card). Our interest is focused mainly on the PCIe bus and the software layer. Fast Internal Bus implemented directly in the FPGA is used for data transmissions between DMA buffers and the PCIe bus. The whole architecture of the NetCOPE on the acceleration card with connected verification core is demonstrated in Figure 7.1.

The verification core can use two interfaces: input and output FrameLink for data transfer (FrameLink RX and FrameLink TX) or MI32 interface for transfer of control information. The FrameLink protocol and the MI32 protocol are described in detail in the following text. In Figure 7.1 there are two FrameLink Watches. The FrameLink Watch is a special hardware unit for checking the number of transactions received from or sent to the verification core and it is used for debugging.

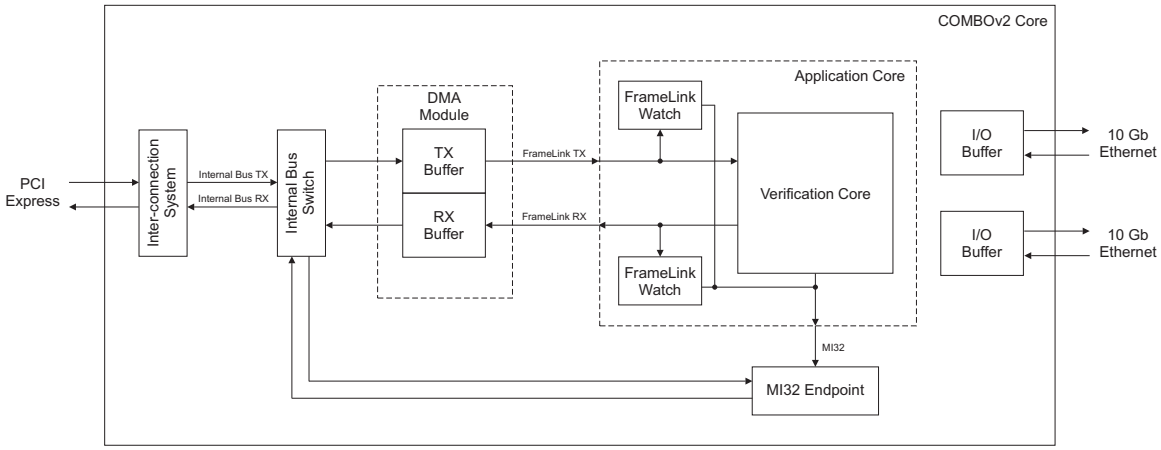


Figure 7.1: The architecture of the hardware part of the NetCOPE platform with connected verification core.

7.1.1 Data Transfers

Transfer of data between the RAM memory and the acceleration card is realized through ring buffers in NetCOPE. The first pair of ring buffers is located directly in the FPGA (DMA TX and RX buffers in Figure 7.1) and similarly the second pair of buffers is located in system kernel memory space in the RAM. A data transfer between software and hardware buffers is managed by four pointers: one pair of pointers defines the start (*StartPointer*) and the end (*EndPointer*) of a continuous data block. These four pointers are controlled by DMA Controllers and DMA transfers are initialized according to their values.

The main principles describing modification of pointers are apparent from the following text (we simplify the description by assuming a pointer arithmetic modulo size of a buffer):

- Transmission of data from a TX RAM buffer to a TX DMA buffer (Figure 7.2).

  1. Software application inserts new data of size *NewDataSize* to the TX RAM buffer and informs the relevant DMA Controller about this event by incrementing the value of *SWEndPointer*.
  2. The DMA Controller checks if there is enough space in the corresponding RX DMA buffer ($HWStartPointer - HWEndPointer \geq NewDataSize$). In the positive case a new DMA transfer is initialized, in the negative case the DMA Controller postpones the transfer.
  3. After the transmission of data to the TX DMA buffer, the DMA Controller increments the value of *SWStartPointer* (the data are released from the TX RAM buffer) and also the value of *HWEndPointer* (the TX DMA buffer is informed about the size of the received data).
  4. When the data are sent from the TX DMA buffer to hardware units, the DMA Controller increments the value of *HWStartPointer*, thus releasing the data from the TX DMA buffer.

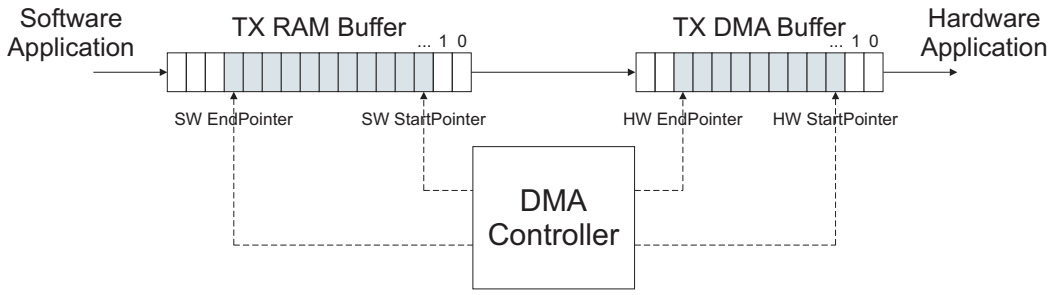


Figure 7.2: Transmission of data from a TX RAM buffer to a TX DMA buffer.

- Transmission of data from a RX DMA buffer to a RX RAM buffer (Figure 7.3).

  1. Hardware application inserts new data of size *NewDataSize* to the RX DMA buffer and informs the relevant DMA Controller about this event by incrementing the value of *HWEndPointer*.
  2. The DMA Controller checks if there is enough space in the corresponding RX RAM buffer ($SWStartPointer - SWEndPointer \geq NewDataSize$). If there is, a new DMA transfer is initialized, if there is not, the input data is delayed.
  3. After the transmission of data to the RX RAM buffer, the DMA Controller increments the value of *HWStartPointer* (the data are released from the RX DMA buffer) and also the value of *SWEndPointer* (the RX RAM buffer is informed about the size of the received data).
  4. If data are taken out from the RX RAM buffer by a software application, the DMA Controller increments the value of *SWStartPointer*, thus releasing the data from the RX RAM buffer.

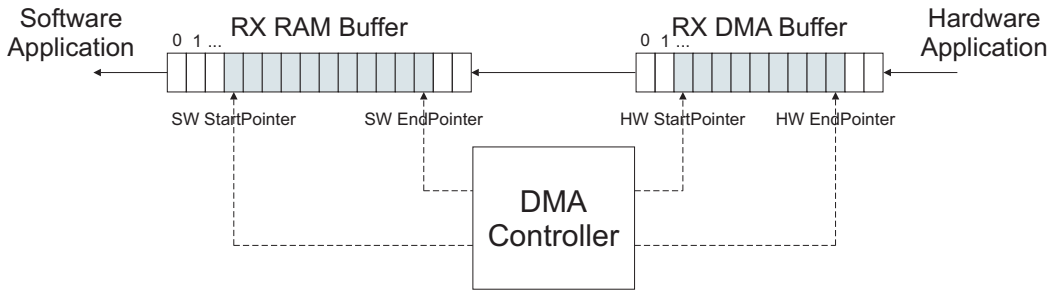


Figure 7.3: Transmission of data from a RX DMA buffer to a RX RAM buffer.

7.2 FrameLink

FrameLink is a packet-oriented synchronous point-to-point protocol for data transfers designed at Liberouter. It is inspired by the LocalLink protocol of Xilinx [6]. The description of signals of an input FrameLink interface follows (the signals of an output FrameLink interface are the same with reversed direction):

- `DATA` — input signal for data transmission, width is N bits, $N \in \{8, 16, 32, 64, 128\}$.
- `REM` — input signal which defines the number of valid bytes in the last word of `DATA`, `REM` width is $\log_2(N)$ bits and the value of this signal is active only if the `EOP_N` signal is active.
- `SOF_N` — start of frame, this 1-bit input signal defines the beginning of a data packet which can by composed of more parts bounded by the `SOP_N` and the `EOP_N` signals. `SOF_N` is always set at the same clock cycle as the `SOP_N` signal of the first part of the data packet.
- `EOF_N` — end of frame, this 1-bit input signal defines the end of the data packet, it is always set at the same clock cycle as the `EOP_N` signal of the last part of the data packet.
- `SOP_N` — start of part, this 1-bit input signal defines the beginning of one part of the data packet.
- `EOP_N` — end of part, this 1-bit input signal defines the end of one part of the data packet.
- `SRC_RDY_N` — 1-bit input signal which determines the readiness of the source side to send new data. An active value of this signal indicates that all above signals have been properly set.
- `DST_RDY_N` — 1-bit output signal which determines the readiness of the destination side to receive new data. An active value of this signal together with an active value of `SRC_RDY_N` indicates that the destination side has accepted data from the source.

Control signals of FrameLink (`SOF_N, EOF_N, SOP_N, EOP_N, SRC_RDY_N, DST_RDY_N`) are all active in logical 0. We also assume that the sending and the receiving party of one FrameLink point-to-point connection share the same clock and `RESET` signal (which is active in logical 1).

According to the dependence of signals in FrameLink the following set of assertions for a FrameLink interface emerges:

1. The `SRC_RDY_N` signal may be active only if the `RESET` signal is inactive.
2. The `SOF_N` signal may be active only if the `SOP_N` signal is active.
3. The `EOF_N` signal may be active only if the `EOP_N` signal is active.
4. After the `EOP_N` signal is active, data cannot be sent (i.e. `SRC_RDY_N` and `DST_RDY_N` cannot be both active), until `SOP_N` is active.
5. Each active `SOP_N` must be, after some time, followed by an active `EOP_N`.
6. Each active `SOF_N` must be, after some time, followed by an active `EOF_N`.

7.3 MI32

MI32 is a special protocol mainly used for transfer of control information. It is primarily designed for communication with components that map their configuration registers into the memory address space of the acceleration card. Typically, MI32 is used for initialization of hardware components, and for reading or writing control information from or to the component's memory. The description of interface signals of the MI32 protocol follows:

- `ADDR` — input signal for address, address width is 32 bits and is associated with one position of hardware component's memory address space.
- `DWR` — input signal for data that are subsequently written in the memory address space on address defined by the `ADDR` signal, provided that the `WR` signal is active. Data width is 32 bits.
- `DRD` — output signal for data that are read from the memory address space from address defined by the `ADDR` signal, provided that the `RD` signal is active. Data width is 32 bits.
- `BE` — 4-bit input signal with a bitmap that defines the validity of transmitted bytes in a `DWR` word.
- `RD` — 1-bit input signal that specifies read request from hardware component's memory address space.
- `WR` — 1-bit input signal that specifies write request to hardware component's memory address space.
- `ARDY` — 1-bit output signal that signalizes validity of address on the `ADDR` signal.
- `DRDY` — 1-bit output signal that signalizes validity of data on the `DRD` signal.

Control signals of MI32 (`RD, WR, ARDY, DRDY`) are active in logical 1. We also assume that the sending and the receiving party of one MI32 point-to-point connection share the same clock and `RESET` signal (which is active in logical 1).

According to the dependence of signals in the MI32 protocol the following set of assertions for a MI32 interface emerges:

1. The `RD` signal may be active only if the `RESET` signal is inactive.
2. The `WR` signal may be active only if the `RESET` signal is inactive.
3. The `ARDY` signal must be active together with the `RD` signal or the `WR` signal.
4. The `WR` signal cannot be active together with the `RD` signal.

7.4 Software Layer

Apart from low-level drivers NetCOPE offers a software library called libsze2 implemented in the C language that allows the user to comfortably and effectively implement applications working with an acceleration card. The main concept used in this library allows achieving the optimal data transfer speed by eliminating data copies between software memory and application in hardware (usually done by a processor). Also the name of the library addresses this issue (it is an abbreviation of *straight zero copy data version 2*). Data are prepared directly in the memory and transferred to the card through DMA channels. One `sze` interface corresponds to one DMA channel in the FPGA.

In the implementation of the accelerated version of the framework we use the following functions of the libsze2 library:

- `szedata_open` — defines connection to a hardware device and after initialization returns a pointer to a structure of type `szedata`, which describes the device (handle) and is passed as a parameter in all following functions.
- `szedata_subscribe` — subscribes (i.e. attaches) to requested interfaces.
- `szedata_start` — activates interfaces, after a call to this function it is possible to send or receive data through an activated interface.
- `szedata_prepare_packet` — prepares a `szedata` packet and stores it into a preallocated space in a TX RAM buffer.
- `szedata_try_write_next` — tries to send a `szedata` packet through a `sze` interface; if a call to this function fails, e.g. because the buffers are full, it can be tried again later.
- `szedata_read_next` — reads one `szedata` packet from a `sze` interface.
- `szedata_close` — deactivates interfaces, flushes buffers.

7.5 Direct Programming Interface

As mentioned in the previous section, the NetCOPE provides functions for data transfer to or from an acceleration card (via the libsze2 library). They are written in the C language and it is not possible to call them directly from the software verification environment written in SystemVerilog. However, there is a possibility to build a special interface between SystemVerilog and a foreign programming language (though nowadays only the C language is supported) called the *direct programming interface* (DPI). The main principles of the DPI are described in the following text.

Both sides of the DPI are fully isolated. The separation between the SystemVerilog code and the foreign language is based on the use of functions as natural encapsulation unit in SystemVerilog, so the DPI supports direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as *imported functions*. SystemVerilog functions that are called from a foreign code shall be specified as *exported functions*. The DPI allows for passing data between the two domains through function arguments and results. SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction.

The foreign language layer of the interface specifies how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific data types are represented, and how they are translated to and from predefined foreign language types. Users are responsible for specifying the native types equivalent to the SystemVerilog types used in imported or exported declarations in their foreign code.

The memory spaces owned and allocated by the foreign code and the SystemVerilog code are disjoint. Each side is responsible for its own allocated memory.

7.6 Transfer Protocol Stack Implementation

For every abstract layer of the transfer protocol stack proposed in Section 6.3 we can define independent protocols. For our framework implementation we decided to use the following set of protocols:

NetCOPE protocol — protocol of the Endpoint Layer, identifies a destination point for data transfer which is realized through DMA channels of the NetCOPE application core. The protocol adds the NetCOPE header at the start of transmitted data. The size of the NetCOPE header is 4 bytes and contains three items:

- NetCOPE Endpoint ID [1 byte] — identifier of the destination unit,
- NetCOPE Endpoint Protocol [1 byte] — output interface protocol of the destination unit,
- Reserved [2 bytes] — reserved space for future extension.

Transaction protocol — protocol of the Transaction Layer, defines types of transmitted transactions between hardware and software. There are two basic types of transactions: data and control. The Transaction protocol adds the Transaction header, the size of which is 2 bytes and contains two items:

- Transaction Type [1 byte] — allows to distinguish data and control transactions and also the type of a control transactions,
- Reserved [1 byte] — reserved space for future extension.

For verification purposes we define the following six types of transactions:

- data transaction — identifies transmitted data, Transaction Type is set to 0,
- control transaction `start` — instruction for starting activity of the addressed unit, Transaction Type is set to 1,
- control transaction `wait` — instruction for waiting for the number of clock cycles defined in the data part, Transaction Type is set to 2,
- control transaction `waitforever` — instruction for waiting until the activity of the addressed unit is terminated, Transaction Type is set to 3,
- control transaction `stop` — instruction for terminating activity of the addressed unit, Transaction Type is set to 4,
- control transaction `delay` — instruction which defines the delay between and inside a transaction, Transaction Type is set to 5,

Interface protocol — identifies a particular physical or virtual interface as the destination of a routed transaction. This enables to differentiate between interfaces that belong to the same protocol. The Interface protocol adds the Interface header, the size of which is 2 bytes, that contains the following two items:

- Interface ID [1 byte] — identifier of the interface,
- Reserved [1 byte] — reserved space for specific protocol information.

Figure 7.4 demonstrates concatenation of protocol headers to transmitted data packets at each layer of the transfer protocol stack.

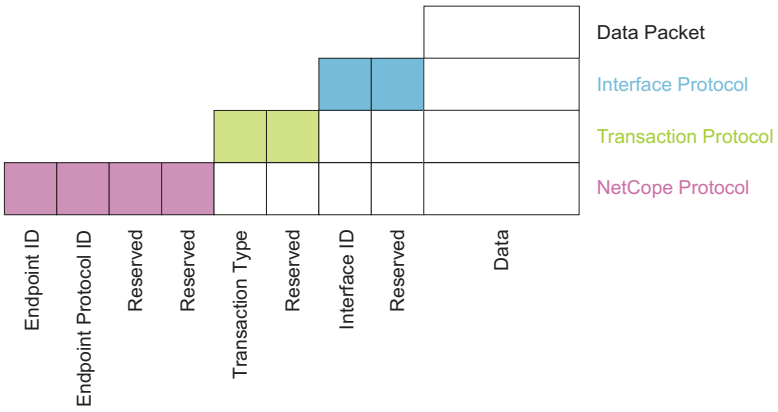


Figure 7.4: Encapsulation of a data packet through stack layers.

7.7 Maintaining Reproducibility of Verification

Our goal is to obtain the same progress of verification runs in the non-accelerated and the accelerated version of the framework. The reason for this is self-evident: when a bug occurs in the accelerated version, it is possible to run the non-accelerated version and explore the origin of the bug in detail in the perfect debugging environment of a simulator.

A few steps have to be taken to achieve this goal:

1. The transaction generator in the testbench must be the same for the non-accelerated and the accelerated version of the verification framework. We picked *the pseudo-random number generator* (PRNG) for its beneficial properties for verification, as it produces a sequence of numbers that approximates the properties of random numbers. The generation starts after initialization from a seed state and always produces the same sequence when initialized with the same seed, which facilitates debugging in verification. The maximum length of the generated sequence before it begins to repeat is determined by the size of the state measured in bits. However, since the length of the maximum period potentially doubles with each bit, it is easy to build PRNGs with periods long enough for many practical applications. We can find such a generator directly in SystemVerilog.

2. In hardware verification environment we need two clock domains to get the same time behaviour of the verification process. This is because the following may happen: either drivers do not contain enough prepared data for the DUT, or monitors cannot send data to the software because the channel to the software is full. As this would lead to invalid data sent to the DUT in the former case or dropping of received data in the latter case, such a verification run would not be reproducible in the software version (and neither in the accelerated version). To guarantee reproducibility of a verification run in the hardware environment, we propose the solution to turn off the clock signal of the DUT. This can be done using the CLOCK GATE (which maps to the BUFGCE component on the Virtex 5 FPGA). However, gating of the clock signal of the DUT can lead to a phase shift, therefore it is necessary to implement clock domain crossing between the clock domain of the DUT and the clock domain of the rest of the hardware verification environment. Asynchronous FIFO components can be used on each side of the communication interface of the DUT (in every hardware driver and hardware monitor) to address this issue. This concept is illustratively represented in Figure 7.5.

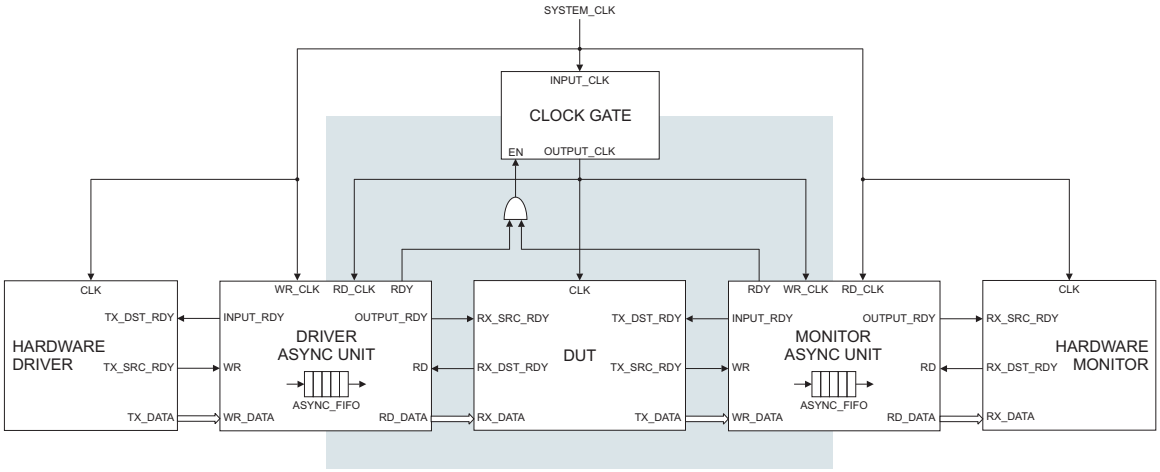


Figure 7.5: Clock domain crossing in the hardware verification environment.

3. Some interfaces may be used only for the initialization of the verified hardware design. The initialization is done usually at the beginning of the device's activity through a special control interface (e.g. MI32). Afterwards, the activity of the interface controller should be paused. In the non-accelerated version, the interface controller only waits for a non-specified number of clock cycles until its activity is completely stopped. To achieve the same behaviour in the hardware verification environment in the accelerated version, the interface controller (Hardware Driver) receives a special control transaction of the Transaction protocol labelled as `waitforever`. In this case the hardware interface controller does not set any signals, but must set signal `RDY` for the CLOCK GATE, otherwise the whole verification run would be stopped.

4. Sometimes it is necessary to deliberately pause the activity of an interface controller. The behaviour is the same as in the previous paragraph, but the activity of the interface controller is interrupted only for a specified number of clock cycles. Again it is required to propagate the information to the hardware and this can be accomplished

through a special control transaction of the Transaction protocol labelled as `wait` (with the defined number of clock cycles for waiting).

5. For verification purposes it is sometimes useful to insert delays between or inside data transactions. These delays can be easily expressed by the input signal that represents the readiness of the source side (in this case the verification testbench) to send data (similarly as the `SRC_RDY_N` signal in FrameLink). For the same behaviour of the non-accelerated and the accelerated version of the verification, it is necessary to maintain the same number and range of delays. We achieve this by a propagation of software delays to the hardware through a special control transaction of the Transaction protocol labelled as `delay`.

6. For the same time behaviour of both framework versions, it is sometimes necessary to set identically the signal representing the readiness of the destination side to accept new data (similarly as the `DST_RDY_N` signal in FrameLink). We can achieve this by connecting PRNGs to drive such a signal in the non-accelerated and also in the accelerated version of the framework. They are initialized with the same seed, so the same time behaviour of the signal is guaranteed.

7. In both versions there are special instructions for starting and terminating the activity of interface controllers. Software units are simply enabled or disabled by special function calls. The activity of hardware units is driven through special control transactions of the Transaction protocol labelled as `start` and `stop`.

7.8 Verification Packages

The architecture of the framework complies to the principles of popular verification methodologies so the level of understandability for verification engineers remains the same. Software verification environments designed in SystemVerilog can be easily modified or extended to enable acceleration. It is also possible to create a completely new verification environment. Only two steps are required: framework components need to be connected to the verification environment and a testcase created according to the template.

Every component in the verification environment performs a special function. The user can connect her own components or components prepared in packages. It is also possible to extend components through inheritance. For implementation purposes we created packages of basic software and hardware components and also packages for interface-oriented components, such as the FrameLink package or the MI32 package. Of course, this set of packages is fully extensible.

Chapter 8

Bug Hunting

If the time behaviour of signals on interfaces is not correct according to the specification, or when some unexpected transaction occurs on an output interface of the verified design, such a situation can be marked as a failing scenario.

Verification engineers usually prefer software verification environments to explore origins of failures because they make use of a well-designed debugging environment of a simulator. The simulator offers a lot of advanced debugging functions like monitoring signal changes in time (in the form of a waveform) or detailed analysis of assertions and coverage. The non-accelerated version of the designed framework introduces exactly such a type of verification environment with all its benefits. But as mentioned before, this type of verification environment is suitable especially when debugging basic system functions with a small number of input transactions (the typical number being between 1,000 and 10,000). This is because the rising complexity of verified hardware designs causes an increase of the time of simulation and also higher requirements for storage of a detailed simulation run in the memory. This issue was the main reason why the accelerated version of the framework was designed. In the accelerated version of the verification environment we can verify complex designs very quickly and with much higher number of transactions (1,000,000 and more). The next benefit is testing directly in the hardware environment. But using this approach, we lose the opportunity to see detailed time behaviour of the verification process in the simulation. The output of the accelerated version informs only about the result of the verification, if it ended correctly or not. If a bug occurs, the verification reports the number of the transaction where the bug caused inconsistency between the received and the expected transaction in the Scoreboard. In order to obtain more detailed information about the error that caused the mismatch of the received transaction and the transaction in the Scoreboard, it is possible to apply approaches to monitor the time behaviour of the verification process directly in the hardware and also directly explore the reason of the failure. An example of such an approach would be adding a timestamp to every transaction. If a bug occurs, each transaction in the hardware can be identified according to the timestamp and moreover transactions in the neighbourhood of this transaction can be easily identified too. Subsequently the verification run can be started again while monitoring the time behaviour during processing of selected transactions with e.g. the ChipScope Pro Analyzer [1].

The ChipScope Pro tool inserts a logical analyzer, a system analyzer, and virtual I/O low-profile software cores directly into a software design and allows to view any internal signal or a node, including embedded hard or soft processors. The signals are captured in the system at the speed of operation and displayed and analyzed using the ChipScope Pro Analyzer tool.

8.1 Analysis of a Failing Scenario

After an error is detected in the accelerated version of the framework, there are two possibilities how to debug it. The easier way is to start the non-accelerated version of the framework and use the debugging environment in the simulator. But if the bug occurs after many transactions, it is unreal to debug it in the simulation because of the abovementioned reasons. In this case the ChipScope Pro Analyzer or any other hardware monitor can be used.

The following example demonstrates these principles in a few steps when debugging a failing scenario during the verification of the FrameLink FIFO component.

Step 1. A failure is typically expressed by a discrepancy in the transaction table (in the Scoreboard), where all transactions we expect on the output of the verified component are stored. If such a discrepancy occurs, the bug is reported and the verification process is stopped. An example of a similar failing scenario took place also during the verification of the FrameLink FIFO component. The following Figure 8.1 illustrates the detection of the failure in the accelerated version of the framework. The figure shows that the Output Controller received an unexpected transaction from the hardware. The reason was that the size of the expected and the received transaction differed. According to the information printed out it can be easily identified that not only the sizes differed and that the data of the transactions did not match, too. As a result of this failing scenario the verification process was stopped and the state of the transaction table was printed. The verification run failed on the 113th transaction received from the hardware.

```
############ TEST CASE 1 ############
#
# START TIME: Tue May 17 23:20:30 CEST 2011
# OPENING CHANNEL
#
# Size doesn't match!!!!!!!!!!!!
# Unknown transaction received from output controller!
#
# EXPECTED TRANSACTION:
# Part no: 0, Part size: 31, Data:
#    0: c6 85 6a 7d cc 9e 63 34  82 c8 ae 79 8a 4 a3 b3
#   10: 40 7c 92 b5 32 43 db fc  38 d6 a1 c7 bb e5 6b
#
# RECEIVED TRANSACTION:
# Part no: 0, Part size: 30, Data:
# 0000: c6 85 6a 7d cc 9e 63 34  82 c8 ae 79 8a 04 a3 b3
# 0010: bf 11 3f 89 9c 71 6e cc  90 38 7e 88 55 cc
#
# ------------------------------------------------------------
# -- C TRANSACTION TABLE
# ------------------------------------------------------------
# Items added: 2000
# Items removed: 112
#
```

Figure 8.1: Detection of a failure in the accelerated version of the framework.

Step 2. After an error has been detected in the accelerated version of the framework, the non-accelerated version can be used for comfortable debugging (if the bug occurs after a manageable number of transactions).

The debugging environment of a simulator offers more opportunities how to find the source of the failure. One of them is a detailed view of the time behaviour of signals in the form of a waveform. An example is demonstrated in Figure 8.2. From this visualisation the source of the failure is much more obvious. We can easily identify that despite the FIFO is full (signal `FULL` is active), the FIFO still signalizes its readiness to receive new data (signal `RX_DST_RDY_N` is active). Of course, there are much more options how to detect a failure in the simulator, like an assertion analyzer or a coverage analyzer.

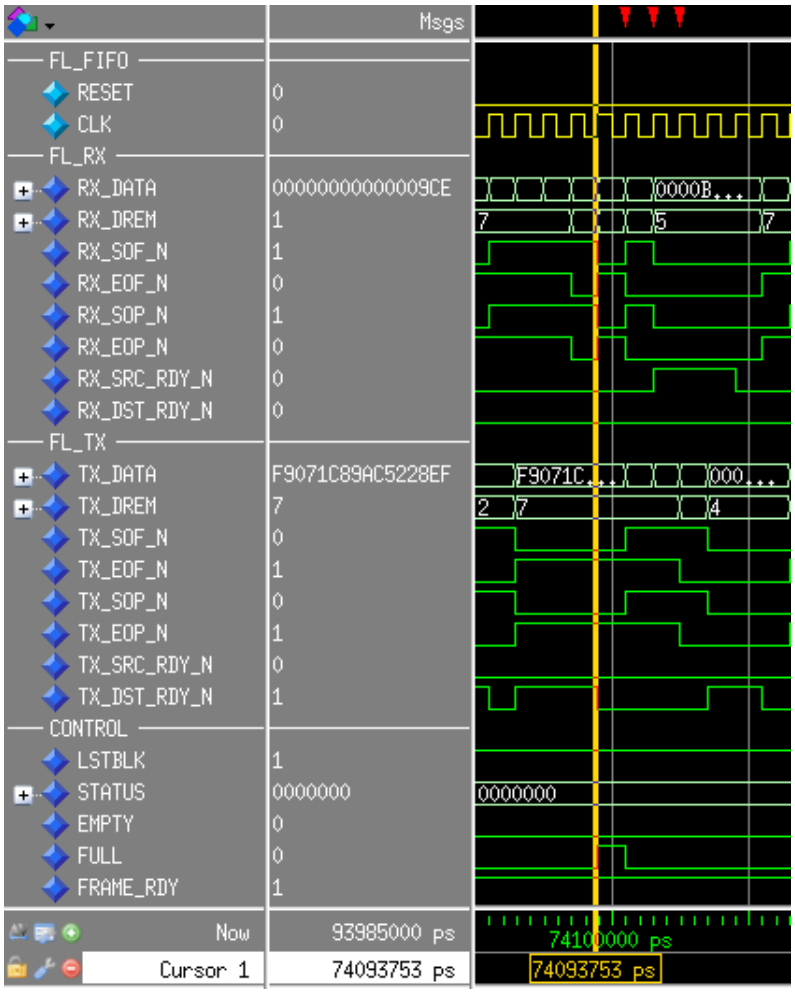


Figure 8.2: A detailed view of the time behaviour of signals in a simulator.

In the assertion analyzer we can check the correctness of the settings of signals on interfaces during the verification process according to the interface protocol specification. If the specification is violated, the failure is reported as an assertion error.

The coverage analyzer allows the user to see how many states of the verified system are checked. Low signal coverage of some interface observed by the simulator indicates

the possibility of the generator not producing enough distinct values on its output.

If the source of the failure is not so obvious it is recommended to monitor the transition of the transaction through the verification environment:

1. The format of the transaction sent to the DUT.
2. The transition of the transaction through the input interface of the DUT using properly designed assertions.
3. The transition of the transaction through the output interface of the DUT using properly designed assertions.
4. The format of the transaction received from the DUT.

In the accelerated version of the framework it is not possible to use the assertion analyzer or the coverage analyzer. Of course, there are approaches how to create them directly in the hardware, e.g. using a state machine implementing a Büchi automaton for expression of assertions in the form of LTL formulae or special hardware units for measuring coverage. But in most cases this is not necessary, because the results of analyses do not change rapidly with the increasing number of input transactions and results from the non-accelerated version of the framework for about 10,000 transactions are according to our experience sufficient enough.

Chapter 9

Experimental Results

We prepared a prototype verification environment for two hardware systems: the FrameLink FIFO component and the Hash Generator. In the following text there is a detailed description of both components followed by the results of the experiments, which were performed on the COMBOv2 LXT155-10G2 acceleration card with the Virtex 5 LX155T FPGA in a PC with a quad-core Intel Xeon E5410@2.33 GHz CPU and 10 GB of RAM. Both components were verified using the proposed verification framework and the experiments demonstrate the difference between the performance of the accelerated and the non-accelerated version of the framework.

9.1 FrameLink FIFO

This component is a *first-in first-out* (FIFO) buffer with two FrameLink interfaces: input (RX) and output (TX). The supported data width of the input and the output interface is 8, 16, 32, 64, or 128 bits. It is possible to set the depth of the FIFO and also the type of the memory to use: either BRAMs (*BlockRAMs*) or LUTs (*look-up tables*). Generic parameters and interface signals are described in Tables 9.1 and 9.2.

For more efficient flow control special signals can be used: `LSTBLK`, `STATUS` and `FRAME_RDY`. After `RESET`, this component reads the first frame on its input and remembers its properties (number of parts). The output `LSTBLK` signal is set to `1` when `BLOCK_SIZE` or less free items are in the FIFO. An additional signal `FRAME_RDY` is set to `1` when at least one whole frame is stored in the FIFO. The `STATUS` signal shows several most significant (or all) bits of the free space counter. This means that the user can get exact information about free items in the FIFO.

Name	Type	Description
`USE_BRAMS`	boolean	Switches between BRAM and LUT memory, `true` value selects BRAM.
`ITEMS`	integer	Number of items that the FIFO can hold.
`BLOCK_SIZE`	integer	The size of a block for the `LSTBLK` signal.
`STATUS_WIDTH`	integer	Width of the `STATUS` signal.

Table 9.1: Generic parameters of the FrameLink FIFO.

Name	Direction	Description
Common Interface		
CLK	in	Clock signal.
RESET	in	Global synchronous reset signal.
FrameLink Interfaces		
RX	inout	Receive interface (write to FIFO).
TX	inout	Transmit interface (read from FIFO).
FIFO Control Interface		
LSTBLK	out	Last block detection.
STATUS	out	MSBs or the exact number of free items in the FIFO.
EMPTY	out	The signal indicating that the FIFO is empty.
FULL	out	The signal indicating that the FIFO is full.
FRAME_RDY	out	The signal indicating that at least one whole frame is in the FIFO.

Table 9.2: The interface of the FrameLink FIFO.

9.1.1 Experiments

It is possible to measure the duration of the verification run using the SystemVerilog `$system()` task. Table 9.3 demonstrates the results of the measurements including the time of the generation of transactions. The table compares the results of the non-accelerated version of the framework (SW) with the results of the accelerated version of the framework (SW-HW). The Acceleration column demonstrates the degree of the achieved acceleration. Table 9.4 and Figure 9.1 show similar results but without the time of the generation of transactions which we measured as 0.5 s for a set of 1,000 transactions. This is because the overhead of the generation of transactions is constant for both approaches and as the overhead is often significant due to applying constraints to the generated random values, it may often be advantageous to store the set of generated transactions into a file and use this file as the input for debugging. Despite the FrameLink FIFO being a very simple component, as can be seen from the summary of consumed resources in the FPGA chip in Table 9.5, the achieved acceleration is over 8 times for 500,000 transactions. The frequency of the design was 125 MHz.

Transactions	SW [s]	SW-HW [s]	Acceleration
10,000	6	6	1.00
50,000	52	32	1.63
100,000	105	60	1.75
200,000	210	116	1.81
500,000	523	282	1.86

Table 9.3: The acceleration achieved in the verification of the FrameLink FIFO including the time of the generation of transactions.

Transactions	SW [s]	SW-HW [s]	Acceleration
10,000	6	6	1.00
50,000	27	7	3.86
100,000	55	10	5.50
200,000	110	16	6.88
500,000	273	32	8.53

Table 9.4: The acceleration achieved in the verification of the FrameLink FIFO without the time of the generation of transactions.

Part of the design	Slices (of 24,320)	Percentage
The FrameLink FIFO	22	0.09 %
The verification core	338	1.39 %
Total	7,576	31.15 %

Table 9.5: Consumed resources in the Virtex 5 FPGA chip for the FrameLink FIFO component.

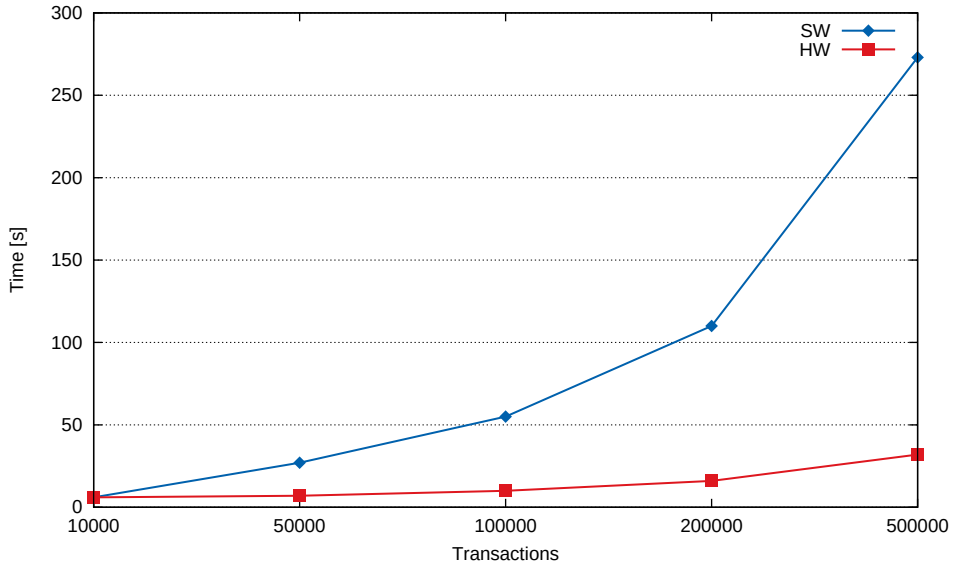


Figure 9.1: The graph of the values from Table 9.4.

9.2 Hash Generator

The Hash Generator computes a hash of selected bytes in a FrameLink packet and inserts it in front of the packet. The desired length of the hash is given by the `FLOWID_WIDTH` generic parameter and supported values are 8, 16, 32, 64, or 128 bits. Which input bytes should be hashed is determined by a bit array that is configurable by the input `MASK` signal. The size of input data is set by the `UH_SIZE` generic parameter. The hash is computed according to the Bob Jenkins Hash Lookup 2 algorithm. Generic parameters and interface signals of the Hash Generator are described in Tables 9.6 and 9.7.

Name	Type	Description
UH_SIZE	integer	The size of a FrameLink packet in bytes. Supported width is 16 and more.
FLOWID_WIDTH	integer	Hash width that must be byte-aligned, 128 bits and more.

Table 9.6: Generic parameters of the Hash Generator.

Name	Direction	Description
Common Interface		
CLK	in	Clock signal.
RESET	in	Global synchronous reset signal.
FrameLink Interfaces		
RX	inout	Receive interface.
TX	inout	Transmit interface.
MI32 Interface		
MI	inout	Interface to software, primarily used for initialization.
Mask Interface		
MASK	in	Mask, each bit corresponds to the one byte of an input FrameLink packet.

Table 9.7: The interface of the Hash Generator.

9.2.1 Experiments

Table 9.8 gives the results of the measurements including the time of the generation of transactions. The table compares the results of the non-accelerated version of the framework (SW) with the results of the accelerated version of the framework (SW-HW). The Acceleration column demonstrates the degree of the achieved acceleration. Table 9.9 and Figure 9.2 show similar results but without the time of the generation of transactions that was measured as 0.6 s for a set of 1,000 transactions. The Hash Generator is a more complex unit as can be seen from the summary of consumed resources in the Virtex 5 FPGA chip in Table 9.10, so the achieved acceleration is much higher up to 15 times for 500,000 transactions. The frequency of the design was 125 MHz.

Transactions	SW [s]	SW-HW [s]	Acceleration
10,000	25	12	2.08
50,000	126	40	3.15
100,000	252	75	3.36
200,000	502	145	3.46
500,000	1,265	364	3.48

Table 9.8: The acceleration achieved in the verification of the Hash Generator including the time of the generation of transactions.

Transactions	SW [s]	SW-HW [s]	Acceleration
10,000	19	6	3.17
50,000	96	10	9.60
100,000	192	15	12.80
200,000	382	25	15.28
500,000	965	64	15.07

Table 9.9: The acceleration achieved in the verification of the Hash Generator without the time of the generation of transactions.

Part of the design	Slices (of 24,320)	Percentage
The Hash Generator	685	2.82 %
The verification core	736	3.03 %
Total	7,992	32.86 %

Table 9.10: Consumed resources in the Virtex 5 FPGA chip for the Hash Generator component.

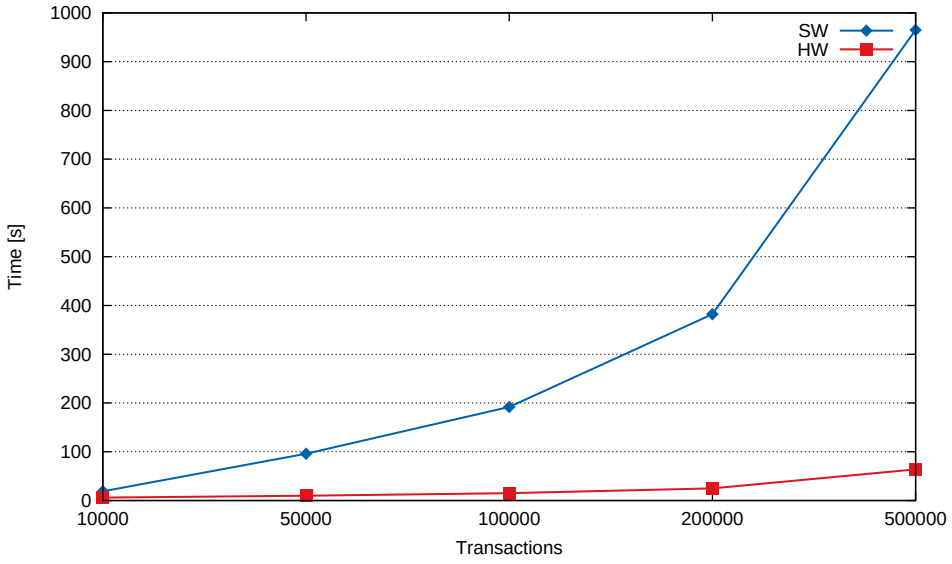


Figure 9.2: The graph of the values from Table 9.9.

In order to fully exploit the capabilities of the accelerated version of the framework it is necessary to verify a complex component. To measure the performance of a complex component while using the previously created verification environments, we built a system with 8 parallel Hash Generators (the Multi-Hash Generator). The summary of consumed FPGA resources is displayed in Table 9.13. This is a synthetic example but demonstrates the possibility to achieve the acceleration of up to 130 times when compared to the non-accelerated version of the framework during a verification of a complex design (that occupies 61 % of the FPGA resources). The detailed results are given in Tables 9.11 and 9.12 and in Figure 9.3. Similarly as in the previous measurements Table 9.11 demonstrates the duration of the verification process including the time of the generation of transactions and Table 9.12 without the time of the generation of transactions. The design ran at 125 MHz.

Transactions	SW [s]	SW-HW [s]	Acceleration
10,000	177	12	14.75
50,000	892	40	22.30
100,000	1,808	75	24.11
200,000	3,536	145	24.38
500,000	8,878	364	24.39

Table 9.11: The acceleration achieved in the verification of the Multi-Hash Generator including time of the generation of transactions.

Transactions	SW [s]	SW-HW [s]	Acceleration
10,000	171	6	28.50
50,000	862	10	86.20
100,000	1,748	15	116.53
200,000	3,416	25	136.64
500,000	8,578	64	134.03

Table 9.12: The acceleration achieved in the verification of the Multi-Hash Generator without the time of the generation of transactions.

Part of the design	Slices (of 24,320)	Percentage
The Multi-Hash Generator	6,841	28.13 %
The verification core	7,242	29.78 %
Total	15,010	61.72 %

Table 9.13: Consumed resources in the Virtex 5 FPGA chip for the Multi-Hash Generator system.

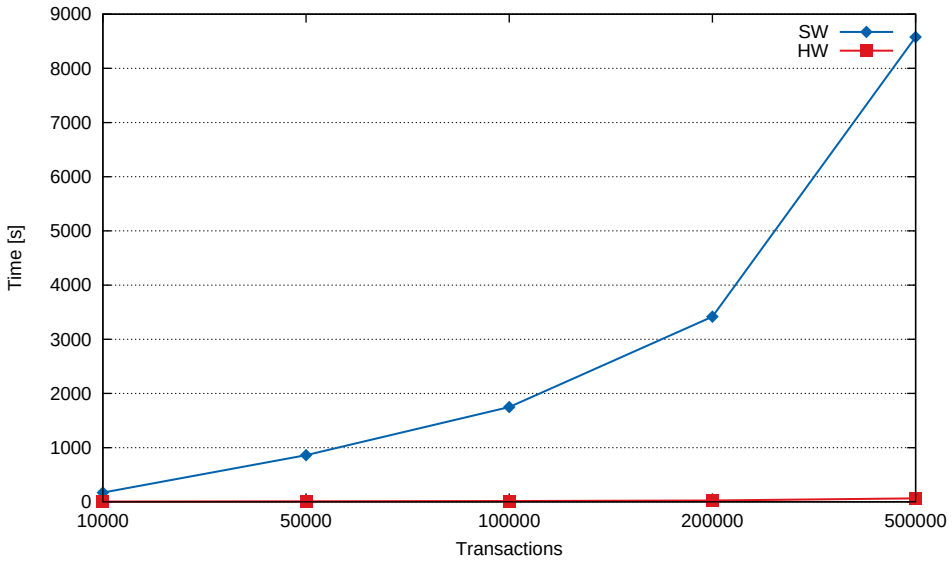


Figure 9.3: The graph of the values from Table 9.12.

Chapter 10

Conclusion

> "The design and testing of an advanced microprocessor chip is among the most complex of all human endeavours."
>
> John Barton (Intel vice-president)

John Barton's words emphasize the idea that the verification of complex hardware systems is not only one of the most important but also one of the most challenging parts in the hardware development process. This is because the checking of all system functions and states according to the specification in the assigned development time is often impossible. This is one of the main reasons why so many verification tools, techniques and approaches have been developed.

Nowadays, verification engineers prefer the following techniques to check system correctness: formal analysis and verification, simulation and testing, and functional verification. The disadvantage of the mentioned approaches is the rising duration of the verification process with the increasing number of input stimuli (which is necessary for sufficient coverage of the state space) and with the increasing complexity of verified hardware components. One of the options how to address this issue and to make the work of verification engineers easier is an acceleration of verification runs.

This work is based on this idea and introduces a framework for hardware acceleration of functional verification written in the SystemVerilog language using the FPGA technology. Current approaches to the acceleration of simulation and functional verification were studied and analyzed and according to the gained knowledge and our own experience two different versions for the implementation of the framework were proposed: the non-accelerated version, which runs in the software and allows to use the debugging environment in a simulator, and the accelerated version, which places the DUT and interface drivers together with the necessary support logic to the FPGA. We strongly believe that the proposed hardware-software strategy (the accelerated version of the framework) is the best decomposition of the task of functional verification between hardware and software with mapping behavioural parts of the testbench to the software and RTL logic to the hardware. The non-accelerated version complements the framework by providing a useful debugging environment.

The proposed framework was tested in the verification of two hardware systems: the FrameLink FIFO and the Hash Generator. The experiments and their results show that using the accelerated version of the framework it is possible to achieve a significant acceleration of over 130 times in the comparison with the run of the verification in the non-accelerated version of the framework.

10.1 Future Work

The implemented prototype of the framework offers a basic functionality for the functional verification purposes either in the accelerated version or in the non-accelerated version. Thanks to the object-oriented nature of the framework it is possible to extend its functions according to the special requirements of verifications engineers, e.g. to include components of popular verification libraries such as OVM or UVM. Also the set of packages can be easily extended. The current version provides only the package of basic verification components and special packages of components working with FrameLink and MI32 interfaces.

The NetCOPE platform, which serves for communication and data transfers between the software and the hardware part of the verification environment in the accelerated version of the framework, could be substituted by an another approach, e.g. using the widely used SCE-MI interface [7].

A more significant acceleration could be achieved by using testbenches (or some components of testbenches) written in the C language because the overhead of some SystemVerilog components is significant. Interesting results could be gained also by mapping all components of the verification environment to the hardware and creating a new verification framework running only in the FPGA. This eliminates the potential bottleneck of the interface between the CPU and the FPGA. However, there are behavioural tasks of testbenches which are not well-suited for synthesis into gate-level. One solution to this issue may be the use of soft-core microprocessor (e.g. MicroBlaze [4] from Xilinx) to carry out such behavioural tasks.

It would also be valuable to compare our approach to the approaches mentioned in Chapter 4, especially with SEmulation [18] and Veloce [5]. We could not perform such a comparison in this work as we do not have an access to these products.

Bibliography

[1] ChipScope Pro. Web pages of Xilinx.
URL: `http://www.xilinx.com/tools/cspro.htm` (May 2011).

[2] Description of COMBO cards. Web pages of Liberouter.
URL: `http://www.liberouter.org/hardware.php` (May 2011).

[3] Liberouter. Web pages of Liberouter. URL: `http://www.liberouter.org` (May 2011).

[4] MicroBlaze Soft Processor Core. Web pages of Xilinx.
URL: `http://www.xilinx.com/tools/microblaze.htm` (May 2011).

[5] Veloce. Web pages of Mentor Graphics.
URL: `http://www.mentor.com/products/fv/emulation-systems/` (May 2011).

[6] Xilinx. Web pages of Xilinx. URL: `http://www.xilinx.com/` (May 2011).

[7] Accellera. *Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual: Version 2.0 Release*, 2007. URL: `http://www.vhdl.org/itc/scemi200.pdf` (May 2011).

[8] Accellera. *Universal Verification Methodology (UVM) 1.0 Class Reference*, 2011.
URL: `http://www.accellera.org/activities/vip/UVM_Class_Reference_Manual_1.0.pdf` (May 2011).

[9] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andrew Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer, 2006. ISBN: 0387-25556-7.

[10] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, 2004. ISBN: 9705394-6-0.

[11] Joytirmoy Daw, Sanjay Gupta, and Suresh Krishnamurthy. Method and System for Hardware Accelerated Verification of Digital Circuit Design and its Testbench. 2004. US Patent no. 7,257,802.

[12] Arthur Freitas. Hardware/Software Co-verification Using the SystemVerilog DPI.
URL: `http://www.qucosa.de/fileadmin/data/qucosa/documents/5410/data/13_Freitas.pdf` (May 2011).

[13] Mark Glasser. *Open Verification Methodology Cookbook*. Springer, 2009.
ISBN: 978-1-4419-0967-1.

[14] IEEE Computer Society. *IEEE Std 1800-2009: IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*, 2009. ISBN: 978-0-7381-6129-7.

[15] Juergen Jaeger. Perform High-speed, Low-cost Prototyping of ASIC Designs. EE Times-India. URL: `http://www.embeddeddesignindia.co.in/STATIC/PDF/200910/EDIOL_2009OCT30_ESL_TA_01.pdf?SOURCES=DOWNLOAD` (May 2011).

[16] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. ISBN: 262-03270-8.

[17] Mark Litterick. FPGA Prototyping in Verification Flows. Verilab, 2006. URL: `http://www.verilab.com/files/fpga_proto.pdf` (May 2011).

[18] Andreas Schwarztrauber. SEmulation: Use your Emulation Board as a Hardware Accelerator for ModelSim SE. *Verification Horizons*, 5:31–34, Dec 2009. URL: `http://www.mentor.com/products/fv/verificationhorizons/upload/horizons-dec-09.pdf` (May 2011).

[19] Aleš Smrčka. *Verification of Asynchronous and Parametrized Hardware designs*. PhD thesis, Faculty of Information Technology, Brno University of Technology, 2010. URL: `http://www.fit.vutbr.cz/research/view_pub.php?id=9435`.